Symbolic Execution in Erlang

Germán Vidal

MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
gvidal@dsic.upv.es

1 Introduction

The concurrent functional language Erlang [1] has a number of distinguishing features, like dynamic typing, concurrency via asynchronous message passing or hot code loading, that make it especially appropriate for distributed, fault-tolerant, soft real-time applications. The success of Erlang is witnessed by the increasing number of its industrial applications. For instance, Erlang has been used to implement Facebook's chat back-end, the mobile application Whatsapp or Twitterfall—a service to view trends and patterns from Twitter—, to name a few. The success of the language, however, also requires the development of powerful testing and verification techniques.

Symbolic execution is at the core of many program analysis and transformation techniques, like partial evaluation, test-case generation or model checking. In this paper, we introduce a symbolic execution technique for Erlang. We discuss how both an overapproximation and an underapproximation of the concrete semantics can be obtained. We illustrate our approach through some examples. To the best of our knowledge, this is the first attempt to formalize symbolic execution in the context of this language, where previous approaches have only considered exploring different schedulings but have not dealt with symbolic data.

2 Erlang Syntax

In this section, we present the basic syntax of a significant subset of Erlang. In particular, we consider a slightly simplified version of the language where some features are excluded (basically, we do not consider modules, exceptions, records, binaries, monitors, ports or process links, most of which are not difficult to deal with but would encumber the notations and definitions of this paper). Nevertheless, this is still a large subset of Erlang and covers its main distinguishing features, like pattern matching, higher-order functions, process creation, message sending and receiving, etc.

The syntax of the language can be found in Figure 1. We denote by $\overline{o_n}$ the sequence of syntactic objects $o_1, \ldots, o_n$. Although Erlang functions are usually defined using patterns and guards, we consider a simpler notation where every function is defined by a single rule with variable parameters and pattern matching is represented by a case expression (similarly to Core Erlang [4]). Nevertheless, every Erlang function could be easily translated to our simpler notation using a case expression as shown in Figure 2.

$$
\begin{array}{rcll}
\mathsf{Program} \ni pgm & ::= & f(\overline{X_n}) \to e. \ \mid \ pgm\ pgm & \\
\mathsf{Exp} \ni e & ::= & bv \ \mid \ [e_1|e_2] \ \mid \ \{\overline{e_n}\} \ \mid \ X \ \mid \ e(\overline{e_n}) & (n \geq 0) \\
& & \mid \ \mathsf{case}\ e\ \mathsf{of}\ cl\ \mathsf{end} \ \mid \ e_1\,!\,e_2 \ \mid \ \mathsf{receive}\ cl\ \mathsf{end} & \\
& & \mid \ p = e \ \mid \ e_1, e_2 & \\
\mathsf{BasicValue} \ni bv & ::= & \mathrm{a} \ \mid \ \mathrm{n} \ \mid \ \mathrm{p} \ \mid \ [\,] \ \mid \ \{\,\} & \\
\mathsf{Value} \ni v & ::= & bv \ \mid \ [v_1|v_2] \ \mid \ \{\overline{v_n}\} & (n > 0) \\
\mathsf{Pattern} \ni p & ::= & bv \ \mid \ X \ \mid \ [p_1|p_2] \ \mid \ \{\overline{p_n}\} & (n > 0) \\
\mathsf{Clauses} \ni cl & ::= & p_1\ \mathsf{when}\ g_1 \to e_1; \ldots; p_n\ \mathsf{when}\ g_n \to e_n & (n > 0)
\end{array}
$$

where $\mathrm{a} \in \mathsf{Atom}$, $\mathrm{n} \in \mathsf{Number}$, $\mathrm{p} \in \mathsf{Pid}$, $X \in \mathsf{Var}$, $g \in \mathsf{Guard}$

Fig. 1. Erlang syntax rules

$$
\left\{ \begin{array}{l} f(\overline{p_n})\ \mathsf{when}\ g_1 \ \to \ e_1; \\ \ldots \\ f(\overline{p'_n})\ \mathsf{when}\ g_m \to \ e_m. \end{array} \right\} \ \Rightarrow \ \left\{ \begin{array}{ll} f(\overline{X_n}) \ \to \ \mathsf{case}\ \{\overline{X_n}\}\ \mathsf{of} & \\ \qquad\qquad \{\overline{p_n}\}\ \mathsf{when}\ g_1 \ \to \ e_1 & \\ \qquad\qquad \ldots & \\ \qquad\qquad \{\overline{p'_n}\}\ \mathsf{when}\ g_m \ \to \ e_m. & \end{array} \right\}
$$

Fig. 2. Translation from Erlang functions to our simpler notation.

Programs are thus sequences of function definitions, where each function f/n is defined by a rule $f(X_1, \ldots, X_n) \to e.$ with $X_1, \ldots, X_n$ distinct variables and the body of the function, e, an expression that might include basic values, lists, tuples, variables, function applications, case expressions, message sending and receiving, pattern matching and sequences.

For simplicity, we do not consider anonymous functions like "$\mathsf{fun}(X_1, \ldots, X_n) \to e\ \mathsf{end}$" and assume that all functions are defined in the program. Extending our developments to anonymous functions would not be difficult and basically requires adding one more rule to the semantics. Besides the functions defined in the program, we consider some of the usual *built-in* functions, e.g.,

- Logical operators: and, not, or, xor.
- Relational operators: $>$, $<$, $=<$, $>=$, $==$, $/=$.
- Arithmetic operators: $+$, $-$, $*$, div.
- Miscellaneous operators: hd (returns the head of a list), tl (returns the tail of a list), element (returns the n-th element of a tuple), $++$ (list concatenation), length (returns the length of a list).
- Concurrent actions. Here, we consider the functions self, that returns the pid of the current process, and spawn, that is used to create new processes. E.g., spawn(foo, [a, 42]) creates a new process that starts calling the function foo(a, 42) and returns the new (fresh) pid assigned to this process.

Only the concurrent actions have side effects. We assume that *guards* can only contain calls to built-in functions without side effects (i.e., all built-ins but self

```
start(N) → S = self(), C = spawn(client, [1, S]), server(N).
server(N) → receive
              {Pid, M} when M < N → Pid ! ok, server(N);
              {Pid, M} when M >= N → Pid ! last
            end.
client(N, Pid) → Pid ! {self(), N},
                 receive Atom → case Atom of
                                  ok → client(N + 1, Pid);
                                  last → ok
                                end
                 end.
```

Fig. 3. Simple client-server example in Erlang

and spawn). In the following, we use capital letters $X, Y, \ldots$ to denote variables, a, b,... to denote atoms, p, p′, ... to denote process identifiers –pids–, $f/n, g/m, \ldots$ to denote functions defined in the program, and *op* to denote a built-in function different from self and spawn. The domain of pids, Pid, and that of atoms, Atom, must be disjoint.

Example 1. Consider the program in Fig. 3 which follows a very simple client-server scheme. Here, the first process is called with $start(N)$, where N is the maximum number of requests accepted by the server. Then, it creates a client (a new concurrent process) and starts the server. Client requests just includes its own pid and the request number. If the request number is smaller than N, the server answers "ok"; otherwise, it answers "last" and terminates. The client keeps asking the server with increasing numbers until it gets the reply "last".

We do not consider I/O in this paper. Therefore, input parameters must be provided through the initial function.

3 Concrete Semantics

The semantics of Erlang is informally described, e.g., in [1]. The past years have witnessed an increasing number of works aimed at defining a formal semantics for the language. Some of the first attempts were done by Huch [11] and, more extensively, by Fredlund [7]. More recent approaches focus on the definition of the distributed aspects of the Erlang semantics, like [5]; this semantics was later refined in [19] and [18], where some assumptions on the future of the language design are proposed. On the other hand, other approaches have formalized the semantics of Erlang by defining its semantics in the framework of rewriting logic [13, 14]. In some cases, Core Erlang [4] is considered, as in [12] and, more recently, in [3].

Unfortunately, there is no commonly accepted semantics and, moreover, most of the above papers only cover part of the language semantics (e.g., [5, 19, 18] are

mainly oriented towards the concurrent features of the language). Therefore, we have recently introduced a semantics for a subset of Erlang in [20]. In the following, we present a more elegant and general version of this semantics that follows some of the ideas in [18] (namely, we make no distinction between the execution of the program in a single node or in a distributed, multi-node environment).

Erlang follows a leftmost innermost operational semantics. Following, e.g., [11, 7], every expression $C[e]$ can be decomposed into a context $C[]$ with a (single) hole and a subexpression e where the next reduction can take place:

$$\begin{array}{rl} C ::= & [] \mid C, e \mid \mathsf{case}\ C\ \mathsf{of}\ cl\ \mathsf{end} \mid C\,!\,e \mid v\,!\,C \mid p = C \mid C(e_1, \ldots, e_n) \\ & \mid f(v_1, \ldots, v_i, C, e_{i+2}, \ldots, e_n) \mid op(v_1, \ldots, v_i, C, e_{i+2}, \ldots, e_n) \\ & \mid [v_1, \ldots, v_i, C|e] \mid \{v_1, \ldots, v_i, C, e_{i+2}, \ldots, e_n\} \end{array}$$

An Erlang *process* is denoted by a tuple $\langle \mathrm{p}; e; q\rangle$, where p is a the process identifier, e is the expression to be evaluated, and q is the process mailbox. An Erlang *system* is a pair $(\Pi, \mathcal{Q})$, where Π is a pool of processes and $\mathcal{Q}$ is the system mailbox (analogous to the *ether* in the semantics of [18]). We assume no order in Π since it is not relevant to our purposes (i.e., we will be interested in exploring *all* possible schedulings within symbolic execution). For implementing actual scheduling policies, an ordering would be required. The system mailbox $\mathcal{Q}$ is a set of triples $(\mathrm{p}, \mathrm{p}', q)$, where q is a list of messages (values) sent from the process with pid p to the process with pid p'. The system mailbox is needed to correctly model a multi-node distributed system (see the discussion in [18]). Basically, Erlang only requires that the messages sent *directly* between two processes must arrive in the same order. However, if the messages follow different paths, say one message is sent directly from p to p'', while another message is sent from p to p'' via p', then there is no guarantee regarding which message arrives first to p''.

The operational semantics is defined by the labelled transition relation $\rightarrow$ shown in Fig. 4. Here, we use the notation $\langle \mathrm{p}; e; q\rangle \ \&\ \Pi$ to denote an arbitrary pool of processes that contains the process $\langle \mathrm{p}; e; q\rangle$. The initial system has the form $(\langle \mathrm{p}_0; e; []\rangle,\ [])$. Most rules are self-explanatory. Let us just explain the more involved ones:

- In rule builtin, we assume a function *eval* that evaluates all built-in's without side effects (i.e., arithmetic or relational expressions, etc).
- In rule fun, we assume that the program *pgm* is a global parameter of the transition system. Moreover, we let $\widehat{e}$ denote a copy of e with local variables renamed with fresh names. The notation $\{\overline{X_n \mapsto v_n}\}$ denotes a substitution binding variables $X_1, \ldots, X_n$ to values $v_1, \ldots, v_n$. The application of a substitution σ to an expression e is denoted by $e\sigma$.
- In rule case, we assume an auxiliary function *match* that takes a value v and the clauses p_1 when $g_1 \rightarrow e_1; \ldots; p_n$ when $g_n \rightarrow e_n$ and returns a pair (e_i, σ) if i is the smaller number such that $p_i\sigma = v$ and $eval(g_i\sigma) = true$.
- The case of rule receive uses a similar auxiliary function *matchrec* that takes a mailbox queue q and the clauses cl, determines the first message v such

$$\textsf{(seq)}\quad \frac{}{(\langle \mathrm{p}; C[v, e]; q\rangle \ \&\ \Pi, \mathcal{Q}) \xrightarrow{\tau} (\langle \mathrm{p}; C[e]; q\rangle \ \&\ \Pi, \mathcal{Q})}$$

$$\textsf{(self)}\quad \frac{}{(\langle \mathrm{p}; C[\textsf{self}()]; q\rangle \ \&\ \Pi, \mathcal{Q}) \xrightarrow{\tau} (\langle \mathrm{p}; C[\mathrm{p}]; q\rangle \ \&\ \Pi, \mathcal{Q})}$$

$$\textsf{(builtin)}\quad \frac{eval(op(\overline{v_n})) = v}{(\langle p; C[op(\overline{v_n})]; q\rangle \ \&\ \Pi, \mathcal{Q}) \xrightarrow{\tau} (\langle p; C[v]; q\rangle \ \&\ \Pi, \mathcal{Q})}$$

$$\textsf{(fun)}\quad \frac{f(\overline{X_n}) \to e. \in pgm}{(\langle \mathrm{p}; C[f(\overline{v_n})]; q\rangle \ \&\ \Pi, \mathcal{Q}) \xrightarrow{\tau} (\langle \mathrm{p}; C[\widehat{e}\{\overline{X_n \mapsto v_n}\}]; q\rangle \ \&\ \Pi, \mathcal{Q})}$$

$$\textsf{(match)}\quad \frac{\exists \sigma.\ p\sigma = v}{(\langle \mathrm{p}; C[p = v]; q\rangle \ \&\ \Pi, \mathcal{Q}) \xrightarrow{\tau} (\langle \mathrm{p}; (C[v])\sigma; q\rangle \ \&\ \Pi, \mathcal{Q})}$$

$$\textsf{(case)}\quad \frac{match(v, cl) = (e, \sigma)}{(\langle \mathrm{p}; C[\textsf{case } v \textsf{ of } cl \textsf{ end}]; q\rangle \ \&\ \Pi, \mathcal{Q}) \xrightarrow{\tau} (\langle \mathrm{p}; (C[e])\sigma; q\rangle \ \&\ \Pi, \mathcal{Q})}$$

$$\textsf{(receive)}\quad \frac{matchrec(q, cl) = (e, \sigma, q')}{(\langle \mathrm{p}; C[\textsf{receive } cl \textsf{ end}]; q\rangle \ \&\ \Pi, \mathcal{Q}) \xrightarrow{\tau} (\langle \mathrm{p}; (C[e])\sigma; q'\rangle \ \&\ \Pi, \mathcal{Q})}$$

$$\textsf{(spawn)}\quad \frac{\mathrm{p}' \text{ is a fresh pid}}{(\langle \mathrm{p}; C[\textsf{spawn}(f, \overline{v_n})]; q\rangle \ \&\ \Pi, \mathcal{Q}) \xrightarrow{\tau} (\langle \mathrm{p}; C[\mathrm{p}']; q\rangle \ \&\ \langle \mathrm{p}', f(\overline{v_n}), [\,]\rangle \ \&\ \Pi, \mathcal{Q})}$$

$$\textsf{(send)}\quad \frac{v_1 = \mathrm{p}' \in Pid \ \wedge\ add\_msg(\mathrm{p}, \mathrm{p}', v_2, \mathcal{Q}) = \mathcal{Q}'}{(\langle \mathrm{p}; C[v_1 \,!\, v_2]; q\rangle \ \&\ \Pi, \mathcal{Q}) \xrightarrow{\tau} (\langle \mathrm{p}; C[v_2]; q\rangle \ \&\ \Pi, \mathcal{Q}')}$$

$$\textsf{(sched)}\quad \frac{(\mathrm{p}, \mathrm{p}') \in sched(\Pi, \mathcal{Q}) \ \wedge\ delivery(\mathrm{p}, \mathrm{p}', \Pi, \mathcal{Q}) = (\Pi', \mathcal{Q}')}{(\Pi, \mathcal{Q}) \xrightarrow{\alpha} (\Pi', \mathcal{Q}')}$$

Fig. 4. Concrete Semantics

that $match(v, cl) = (e, \sigma)$, and returns (e, σ, q'), where q' is obtained from q by deleting message v.

- In rule send, the message is stored in the system mailbox, together with the source and target pids, using the auxiliary function *add_msg*, whose definition is straightforward. Note that the message is not actually delivered to the process with pid p' until the sched rule is applied (see below).
- Finally, rule sched uses the auxiliary function *sched* to model a particular scheduling policy. Basically, it selects two pids $(\mathrm{p}, \mathrm{p}')$ from Π (source and target processes, which might be the same) such that $(\mathrm{p}, \mathrm{p}', q) \in \mathcal{Q}$ and q is not empty. Then, function *delivery* moves the first message of q to the local mailbox of the process with pid p', thus returning a new pair $(\Pi', \mathcal{Q}')$.

Observe that all rules are labeled with τ except for the last one. This is explained by the fact that we are interested in a particular type of computations that we call *normalized* computations. In the following, given a state s, we denote by $s{\downarrow^{\tau}}$ the state that results from s by only applying transitions labeled with τ until no

more transitions labeled with τ are possible, i.e., if $s \equiv s_0 \stackrel{\tau}{\rightarrow} s_1 \stackrel{\tau}{\rightarrow} \ldots \stackrel{\tau}{\rightarrow} s_n \not\stackrel{\tau}{\rightarrow}$, then $s{\downarrow^\tau} = s_n$. Therefore,

Definition 1 (normalized computation). *Let s_0 be the initial system. Then, we say that a computation is* normalized *if it has the form*

$$s_0 \stackrel{\tau^*}{\rightarrow} s_0{\downarrow^\tau} \stackrel{\alpha}{\rightarrow} s_1 \stackrel{\tau^*}{\rightarrow} s_1{\downarrow^\tau} \stackrel{\alpha}{\rightarrow} s_2 \stackrel{\tau^*}{\rightarrow} s_2{\downarrow^\tau} \stackrel{\alpha}{\rightarrow} s_3 \ldots$$

Observe that normalized computations are deterministic since only one rule of the semantics is applicable to any Erlang system.

In the following, we only consider normalized computations in order to reduce the search space. This is similar to hiding the *internal* computations in [10] or using the notion of macro-step semantics of [16], where the execution of a process from a receive statement up to the next receive statement takes place consecutively without the intervention of the scheduler. Of course, this is only fair as long as no process can run infinitely without the execution of a receive statement. This is, however, quite a reasonable restriction for Erlang programs.

Our concrete semantics can be seen as a generalization of that in [18] though we consider the semantics of expressions (while [18] focuses on concurrent actions), and we ignore a number of concurrent actions like process links, monitors, process termination, etc. Note that our semantics could be trivially extended to also consider the loss of messages by just adding a new rule that deletes arbitrary messages from the system mailbox.

Example 2. Consider again the program of Example 1. A computation with this program is shown in Figure 5, where the expression selected for reduction is underlined.

4 Symbolic Execution Semantics

In this section, we introduce a symbolic execution semantics for Erlang. Firstly, one could consider the semantics in Fig. 4 and just define a function *sched* that returns all feasible combinations of processes in the considered system. This is useful to explore all possible schedulings and detect errors (e.g., deadlocks) that only occur in a particular scheduling. This is the aim, e.g., of the model checker McErlang [8]. Basically, McErlang is today a mature tool that combines the use of random test cases (using, e.g., a tool like QuickCheck [2]) with a semantics that explores possible schedulings.

Here, we plan to also cope with missing input data (analogously to the tool Java Pathfinder [15] for model checking of Java bytecode). Reasonably, we assume that the input parameters to an Erlang program must be *values*, i.e., we can call the program of Example 1 with $start(3)$, but calling $start(3 + 4)$ or $start(foo(3, 4))$ is not allowed. Therefore, our *symbolic values* are syntactically equivalent to patterns, i.e., values possibly with variables denoting missing information. The first extension is to redefine the way an expression is decomposed

$$
\begin{array}{ll}
 & (\langle \mathrm{p}_0; \mathit{start}(1); [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; S = \underline{\mathsf{self}()}, C = \mathsf{spawn}(\mathit{client}, [1, S]), \mathit{server}(1); [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \underline{S = \mathrm{p}_0}, C = \mathsf{spawn}(\mathit{client}, [1, S]), \mathit{server}(1); [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \underline{\mathrm{p}_0}, C = \mathsf{spawn}(\mathit{client}, [1, \mathrm{p}_0]), \mathit{server}(1); [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; C = \underline{\mathsf{spawn}(\mathit{client}, [1, \mathrm{p}_0])}, \mathit{server}(1); [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \underline{C = \mathrm{p}_1}, \mathit{server}(1); [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathit{client}(1, \mathrm{p}_0); [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \underline{\mathrm{p}_1}, \mathit{server}(1); [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathit{client}(1, \mathrm{p}_0); [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \underline{\mathit{server}(1)}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathit{client}(1, \mathrm{p}_0); [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \underline{\mathit{client}(1, \mathrm{p}_0)}; [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathrm{p}_0 \,!\, \{\underline{\mathsf{self}()}, 1\}, \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \underline{\mathrm{p}_0 \,!\, \{\mathrm{p}_1, 1\}}, \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle,\ [\,]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \underline{\{\mathrm{p}_1, 1\}, \mathsf{receive} \ldots \mathsf{end}}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\{\mathrm{p}_1, 1\}])]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\{\mathrm{p}_1, 1\}])]) \\
\xrightarrow{\alpha} & (\langle \mathrm{p}_0; \underline{\mathsf{receive} \ldots \mathsf{end}}; [\{\mathrm{p}_1, 1\}]\rangle \ \&\ \langle \mathrm{p}_1; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\,])]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \underline{\mathrm{p}_1 \,!\, \mathsf{last}}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\,])]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \mathsf{last}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\,]), (\mathrm{p}_0, \mathrm{p}_1, [\mathsf{last}])]) \\
\xrightarrow{\alpha} & (\langle \mathrm{p}_0; \mathsf{last}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \underline{\mathsf{receive} \ldots \mathsf{end}}; [\mathsf{last}]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\,]), (\mathrm{p}_0, \mathrm{p}_1, [\,])]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \mathsf{last}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \underline{\mathsf{case\ last\ of\ \ldots\ end}}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\,]), (\mathrm{p}_0, \mathrm{p}_1, [\,])]) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \mathsf{last}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathsf{ok}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\,]), (\mathrm{p}_0, \mathrm{p}_1, [\,])])
\end{array}
$$

Fig. 5. Computation for the program of Example 1

into a context and a reducible expression as follows:

$$
\begin{array}{rl}
C ::= & [] \ \mid\ C, e \ \mid\ \mathsf{case}\ C\ \mathsf{of}\ \mathit{cl}\ \mathsf{end} \ \mid\ C \,!\, e \ \mid\ p \,!\, C \ \mid\ p = C \ \mid\ C(e_1, \ldots, e_n) \\
 & \mid\ f(p_1, \ldots, p_i, C, e_{i+2}, \ldots, e_n) \ \mid\ \mathit{op}(p_1, \ldots, v_i, C, e_{i+2}, \ldots, e_n) \\
 & \mid\ [p_1, \ldots, p_i, C | e] \ \mid\ \{p_1, \ldots, p_i, C, e_{i+2}, \ldots, e_n\}
\end{array}
$$

Our *symbolic systems* are now triples of the form $(\Pi, \mathcal{Q}, \mathcal{C})$, where the new element $\mathcal{C}$ is the so called *path constraint* (initialized to *true*). Loosely speaking, $\mathcal{C}$ contains some constraints on the symbolic values that represent the missing input data, such that the system $(\Pi, \mathcal{Q})$ is reachable (using the concrete semantics) when the input data in the initial system satisfies the constraint $\mathcal{C}$.

4.1 An Overapproximation

First, we consider that symbolic execution must *overapproximate* the concrete semantics. This is useful, e.g., in the context of partial evaluation or when a property that holds for *all* states must be verified. The symbolic execution semantics is shown in Fig. 6. Let us briefly explain the main differences w.r.t. the concrete semantics:

– Rules seq and self remain unchanged, the only difference is that some values might contain variables now and, thus, we consider patterns instead.

$$\textsf{(seq)}\quad \frac{}{(\langle \mathrm{p}; C[p,e]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\tau} (\langle \mathrm{p}; C[e]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C})}$$

$$\textsf{(self)}\quad \frac{}{(\langle \mathrm{p}; C[\textsf{self}()]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\tau} (\langle \mathrm{p}; C[\mathrm{p}]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C})}$$

$$\textsf{(builtin1)}\quad \frac{eval(op(\overline{v_n})) = v}{(\langle p; C[op(\overline{v_n})]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\tau} (\langle p; C[v]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C})}$$

$$\textsf{(builtin2)}\quad \frac{\exists i.\ p_i \text{ is not a value},\ X \text{ is a fresh variable}}{(\langle p; C[op(\overline{p_n})]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\tau} (\langle p; C[X]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C} \wedge (X = op(\overline{p_n}))}$$

$$\textsf{(fun)}\quad \frac{f(\overline{X_n}) \to e. \in pgm}{(\langle \mathrm{p}; C[f(\overline{p_n})]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\tau} (\langle \mathrm{p}; C[\widehat{e}\{\overline{X_n \mapsto p_n}\}]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C})}$$

$$\textsf{(match)}\quad \frac{\exists \sigma.\ p_1\sigma = p_2\sigma}{(\langle \mathrm{p}; C[p_1 = p_2]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\tau} (\langle \mathrm{p}; (C[p_2])\sigma; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C})}$$

$$\textsf{(case)}\quad \frac{(e, \sigma, \mathcal{C}') \in unify(\mathcal{C}, p, cl),\ \mathcal{C}'' = \widehat{\sigma} \wedge \mathcal{C}'}{(\langle \mathrm{p}; C[\textsf{case } p \textsf{ of } cl \textsf{ end}]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\tau} (\langle \mathrm{p}; (C[e])\sigma; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C} \wedge \mathcal{C}'')}$$

$$\textsf{(receive)}\quad \frac{(e, \sigma, q', \mathcal{C}') \in unifyrec(\mathcal{C}, q, cl),\ \mathcal{C}'' = \widehat{\sigma} \wedge \mathcal{C}'}{(\langle \mathrm{p}; C[\textsf{receive } cl \textsf{ end}]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\tau} (\langle \mathrm{p}; (C[e])\sigma; q'\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C} \wedge \mathcal{C}'')}$$

$$\textsf{(spawn)}\quad \frac{\mathrm{p}' \text{ is a fresh pid}}{(\langle \mathrm{p}; C[\textsf{spawn}(f, \overline{p_n})]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\tau} (\langle \mathrm{p}; C[\mathrm{p}']; q\rangle \ \&\ \langle \mathrm{p}', f(\overline{p_n}), [\,]\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C})}$$

$$\textsf{(send)}\quad \frac{v = \mathrm{p}' \in Pid \ \wedge\ add\_msg(\mathrm{p}, \mathrm{p}', p, \mathcal{Q}) = \mathcal{Q}'}{(\langle \mathrm{p}; C[v\,!\,p]; q\rangle \ \&\ \Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\tau} (\langle \mathrm{p}; C[p]; q\rangle \ \&\ \Pi, \mathcal{Q}', \mathcal{C})}$$

$$\textsf{(sched)}\quad \frac{(\mathrm{p}, \mathrm{p}') \in sched(\Pi, \mathcal{Q}) \ \wedge\ delivery(\mathrm{p}, \mathrm{p}', \Pi, \mathcal{Q}) = (\Pi', \mathcal{Q}')}{(\Pi, \mathcal{Q}, \mathcal{C}) \xrightarrow{\alpha} (\Pi', \mathcal{Q}', \mathcal{C})}$$

Fig. 6. Symbolic Execution

- Rule builtin considers now two cases: builtin1, which is equivalent to the previous rule in the concrete semantics, and builtin2 that considers the case when some argument is not a value. In the latter case, the built in function cannot be evaluated and we reduce it to a fresh variable and add the corresponding constraint to the system. E.g., given the expression $3 + Y$, we reduce it to a fresh variable X and add the constraint $X = 3 + Y$ to the system constraint.
- Rule fun remains unchanged. Therefore, applications of the form $X(p_1, \ldots, p_n)$ are not considered since it would involve calling every function and built-in of the program to keep the symbolic execution complete, which is not acceptable (though it could be improved by using a *closure analysis*). If such an expression is reached, we give up and stop symbolic execution with a failure.

– Rule match is similar to the original rule in the concrete semantics but replaces matching with unification. Analogously, rules case and receive mainly replaces the auxiliary functions *match* and *matchrev* with *unify* and *unifyrev* where unification replaces matching as follows. Function *unify* takes a constraint $\mathcal{C}$, a pattern p and the clauses p_1 when $g_1 \to e_1; \ldots; p_n$ when $g_n \to e_n$ and returns a triple $(e_i, \sigma, \mathcal{C}')$ for each i such that $p_i\sigma = p\sigma$ (i.e., σ is a *unifier* of p_i and p) and $\mathcal{C} \Rightarrow \neg g_i\sigma$ cannot be proved (i.e., the unsatisfiability of $g_i\sigma$ cannot be proved); here, $\mathcal{C}'$ is the constraint $\mathcal{C} \wedge g_i\sigma$ (when $g_i\sigma$ is different from *true*). Function *unifyrec* proceeds analogously. Note that we also add the computed unifier to the path constraint (where $\widehat{\sigma}$ denotes the equational representation of a substitution σ). This will be required in the next section. The new functions return a *set* since the pattern might unify with more than one clause whose guard is also satisfiable. Note that this strategy is complete but typically not sound since (besides the limitations of the constraint solver) we might follow several paths while the original, concrete semantics only considers the first clause even if a value matches several clauses.
– Rule spawn, analogously to the case of rule fun, does not consider an expression like spawn$(X, [\overline{p_n}])$, which will be considered a failure. A similar situation happens with rule send. Here, we consider the case where the message is a pattern and, thus, might be a variable. However, we do not consider that the pid of the target process is a variable, since it would involve broadcasting the message to all processes to keep the symbolic execution complete, which is not acceptable.
– Finally, as mentioned before, rule sched just considers a scheduling function *sched* that returns all possible combinations in order to explore all feasible schedulings.

As it is common practice, we assume that the system constraint is checked for *unsatisfiability* at every step. When unsatisfiability cannot be proved (either because it is satisfiable or because the constraint solver is not powerful enough), we continue with the symbolic execution (which is complete, but a potential source of unsoundness).

As in the previous case, only *normalized* symbolic executions are considered.

Example 3. Consider again the program of Ex. 1. Now, Fig. 7 shows a normalized symbolic execution starting with an unknown number K of maximum requests.

4.2 An Underapproximation

So far, we have put the emphasis on completeness (i.e., producing an overapproximation of the original Erlang computations). For this purpose, we had to take a number of decisions that make the resulting search space too huge to scale to real world Erlang applications with thousands or millions of processes. Moreover, there are a number of situations in which we have to give up (i.e., variable applications, process spawning with an unknown function or sending a message

$$
\begin{array}{ll}
 & (\langle \mathrm{p}_0; \mathit{start}(K); [\,]\rangle,\ [\,],\ \mathit{true}) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; S = \underline{\mathsf{self}()}, C = \mathsf{spawn}(\mathit{client}, [1, S]), \mathit{server}(K); [\,]\rangle,\ [\,],\ \mathit{true}) \\
\ldots & \\
\xrightarrow{\alpha} & (\langle \mathrm{p}_0; \underline{\mathsf{receive} \ldots \mathsf{end}}; [\{\mathrm{p}_1, 1\}]\rangle \ \&\ \langle \mathrm{p}_1; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\,])],\ \mathit{true}) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \underline{\mathrm{p}_1\ !\ \mathrm{ok}}, \mathit{server}(K); [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\,])],\ 1 < K) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \underline{\mathrm{ok}, \mathit{server}(K)}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\,]), (\mathrm{p}_0, \mathrm{p}_1, [\mathrm{ok}])],\ 1 < K) \\
\xrightarrow{\tau} & (\langle \mathrm{p}_0; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle \ \&\ \langle \mathrm{p}_1; \mathsf{receive} \ldots \mathsf{end}; [\,]\rangle,\ [(\mathrm{p}_1, \mathrm{p}_0, [\,]), (\mathrm{p}_0, \mathrm{p}_1, [\mathrm{ok}])],\ 1 < K) \\
\ldots &
\end{array}
$$

Fig. 7. Partial symbolic execution for the program of Example 1

to an unknown pid) because dealing with them is simply intractable. Moreover, termination might be an issue, and the definition of appropriate subsumption and abstraction operators would also be needed.

As an alternative, we propose in this section a *sound* symbolic execution that computes an *underapproximation* of the concrete semantics. This is useful for many applications (like test case generation or model checking), and it is often more scalable and avoids false positives. Here, we follow the approach of [9, 17] to so called *concolic execution* and consider the following scheme:

- Processes are slightly extended as follows: $\langle \mathrm{p}, e_c, e_s, q\rangle$, where p is a pid, e_c is a concrete expression, e_s is a symbolic expression, and q is the mailbox queue. The concrete expression drives the application of the rules of the symbolic execution semantics, while the symbolic expression is only used to compute the corresponding path constraint.
- Now, one starts the execution with a random test input data and execute the program using basically the symbolic execution semantics of Fig. 6 using an initial system like $\langle \mathrm{p}_0,\ \mathit{start}(1),\ \mathit{start}(K),\ [\,],\ \mathit{true}\rangle$.
- Then, when the computation terminates, we produce a sequence of the form $E_0, E_1, E_2, \ldots, E_n$ where each E_i is either a constraint $\mathcal{C}_i$ (associated to the i-th computation step) or the symbol α denoting one application of the sched rule. We now traverse this sequence starting from the last element and either negate a constraint or consider alternative schedulings, depending on the type of the considered element. In the case of a negated constraint, we use a constraint solver to produce a new set of input data.[1] Either way, a new symbolic execution is considered and the process starts again. Usually, backtracking can be used to explore all possibilities.
- If the algorithm terminates and the constraint solver is always able to generate a new set of input data, concolic execution is both sound and complete; otherwise, it is only sound (an underapproximation). Termination can be ensured using, e.g., a maximum depth for symbolic execution.

[1] Note that, in contrast to the case of standard symbolic execution, the constraint solver should only produce values for the negated constraint rather than for the complete path constraint, which is usually much simpler.

Example 4. Consider again the program of Example 1 and the initial call $start(1)$. The initial system is thus $(\langle \mathrm{p}_0, start(1), start(K), [\,]\rangle,\ [\,],\ true)$. Here, we would basically perform the same computation shown in Example 2 but using the rules of Fig. 6 to also obtain the following sequence of constraints and scheduling steps: $(\alpha,\ 1 >= K)$ (only the constraints relevant to the symbolic input data, K, have to be considered). Now, by negating the constraint $1 >= K$, we produce a new value, e.g., $K = 5$, and consider a new symbolic execution starting from the system $(\langle \mathrm{p}_0, start(5), start(K), [\,]\rangle,\ [\,],\ true)$. Finally, one should consider alternative schedulings (because we reach a symbol α) but no alternative exists. Therefore, we conclude that executing $start(1)$ and $start(5)$ is sufficient to cover all possible execution paths for the source program.

5 Discussion

In this paper, we have explored the definition of a symbolic execution semantics for the functional and concurrent language Erlang. We proposed both an overapproximation and an underapproximation—based on a variant of symbolic execution called concolic execution—. In principle, it seems that the underapproximation will be more practical and scalable in order to design a tool for model checking and/or test case generation. We are only aware of the approach of [6] to symbolic execution in Erlang, though no formalization is introduced in this paper (it is only explained informally). Hence we think that our approach is a promising step towards defining a practical symbolic execution technique for Erlang, which can be used in different contexts like partial evaluation, model checking or test case generation.

References

1. J. Armstrong, R. Virding, and M. Williams. *Concurrent programming in Erlang (2nd edition)*. Prentice Hall, 1996.
2. T. Arts, J. Hughes, J. Johansson, and U.T. Wiger. Testing telecoms software with quviq QuickCheck. In *Proc. of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 2–10. ACM, 2006.
3. R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. A Declarative Debugger for Sequential Erlang Programs. In Margus Veanes and Luca Viganò, editors, *Proc. of the 7th International Conference on Tests and Proofs (TAP 2013)*, Lecture Notes in Computer Science, pages 96–114. Springer, 2013.
4. R. Carlsson. An Introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, 2001. Available from `http://www.erlang.se/workshop/carlsson.ps`.
5. K. Claessen and H. Svensson. A semantics for distributed Erlang. In Konstantinos F. Sagonas and Joe Armstrong, editors, *Proc. of the 2005 ACM SIGPLAN Workshop on Erlang*, pages 78–87. ACM, 2005.
6. Clara Benac Earle. Symbolic program execution using the Erlang verification tool. In María Alpuente, editor, *Proc. of the 9th International Workshop on Functional and Logic Programming (WFLP 2000)*, pages 42–55, 2000.
7. L.-A. Fredlund. *A framework for reasoning about Erlang code*. PhD thesis, The Royal Institute of Technology, Sweeden, 2001.

8. Lars-Ake Fredlund and Hans Svensson. McErlang: a model checker for a distributed functional programming language. In Ralf Hinze and Norman Ramsey, editors, *Proc. of ICFP 2007*, pages 125–136. ACM, 2007.
9. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005.
10. L.H. Haß and T. Noll. Equational abstractions for reducing the state space of rewrite theories. *Electr. Notes Theor. Comput. Sci.*, 238(3):139–154, 2009.
11. Frank Huch. Verification of Erlang Programs using Abstract Interpretation and Model Checking. In Didier Rémi and Peter Lee, editors, *Proc. of ICFP '99*, pages 261–272. ACM, 1999.
12. M.R. Neuhäußer and T. Noll. Abstraction and Model Checking of Core Erlang Programs in Maude. *Electr. Notes Theor. Comput. Sci.*, 176(4):147–163, 2007.
13. Thomas Noll. A Rewriting Logic Implementation of Erlang. *Electr. Notes Theor. Comput. Sci.*, 44(2):206–224, 2001.
14. Thomas Noll. Equational Abstractions for Model Checking Erlang Programs. *Electr. Notes Theor. Comput. Sci.*, 118:145–162, 2005.
15. C.S. Pasareanu, W. Visser, D.H. Bushnell, J. Geldenhuys, P.C. Mehlitz, and Neha Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.
16. K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *Proc. of the 9th Int'l Conference on Fundamental Approaches to Software Engineering (FASE'06)*, pages 339–356. Springer LNCS 3922, 2006.
17. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. of ESEC/SIGSOFT FSE 2005*, pages 263–272. ACM, 2005.
18. H. Svensson, L.-A. Fredlund, and C. Benac Earle. A unified semantics for future Erlang. In *Proc. of the 9th ACM SIGPLAN workshop on Erlang*, pages 23–32. ACM, 2010.
19. Hans Svensson and Lars-Ake Fredlund. A more accurate semantics for distributed Erlang. In Simon J. Thompson and Lars-Ake Fredlund, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang*, pages 43–54. ACM, 2007.
20. G. Vidal. Towards Erlang verification by term rewriting. In *Proc. of the 23th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, pages 161–178. Technical Report TR-11-13, Universidad Complutense de Madrid, 2013. Available from `http://users.dsic.upv.es/~gvidal/`.
21. G. Vidal. Towards symbolic execution in erlang. Technical report, DSIC, UPV, 2014.